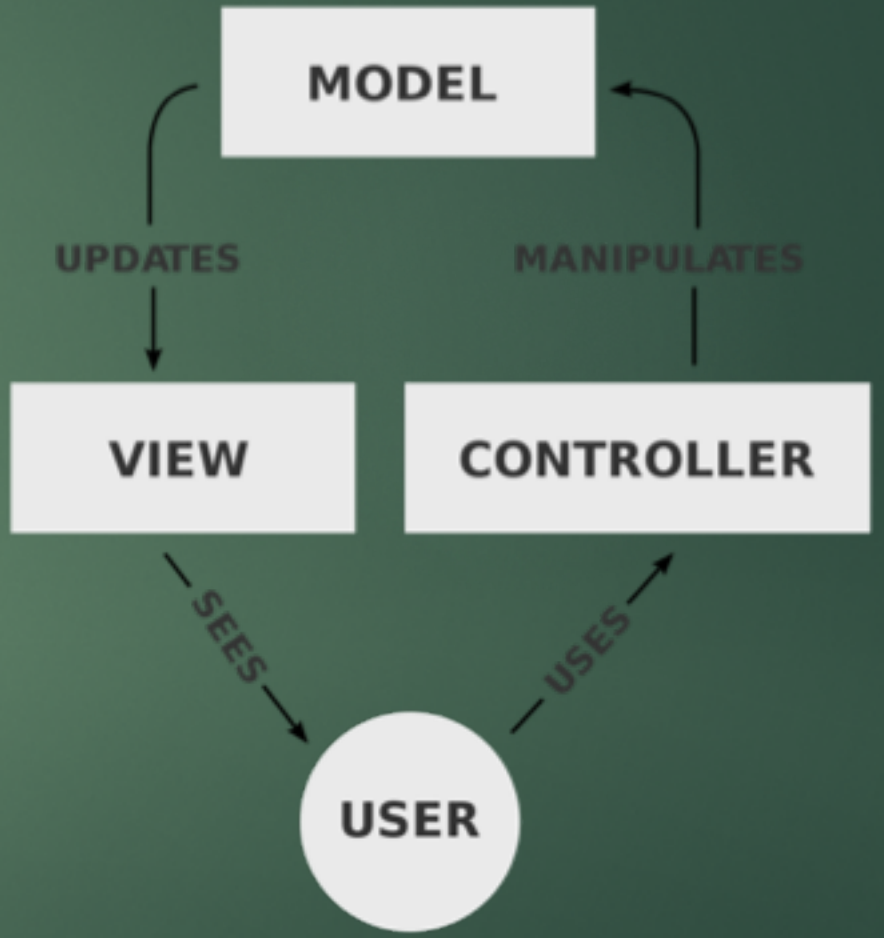
Architectural Patterns

Model-View-Controller

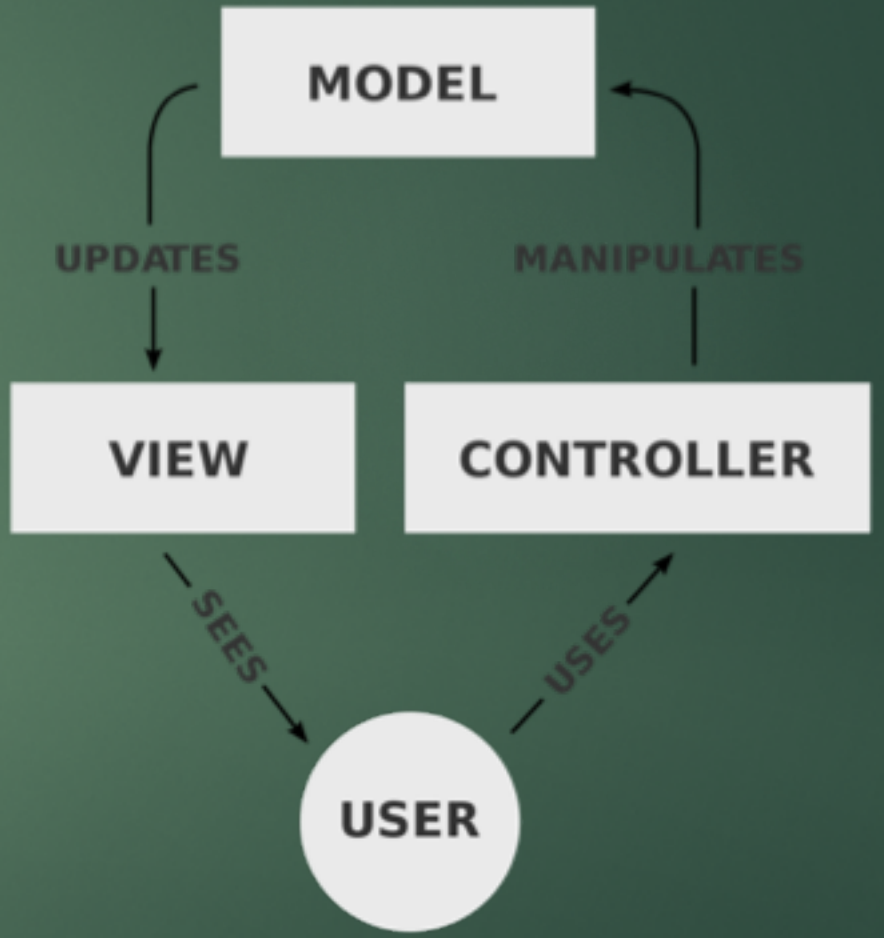
Model-View-Controller

- Divides the application into three components:
  - The model
  - The view
  - The controller
- Separates the internal representation of data from how it's displayed to the user
- Allows more efficient code reuse
- Easier to coordinate development across the app



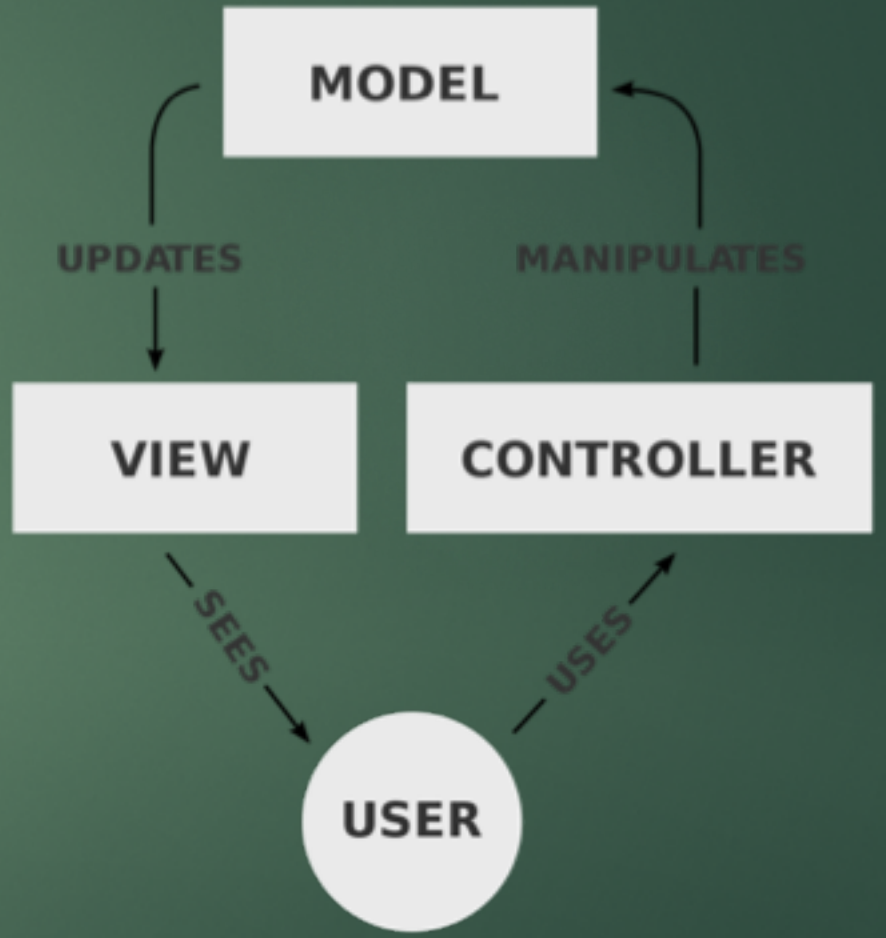
MVC : Model

- Represents all data in the app
  - Any data retrieved from services
  - Any data collected from or modified by the user
- Also handles business logic / application rules
- Receives user input via the *controller*



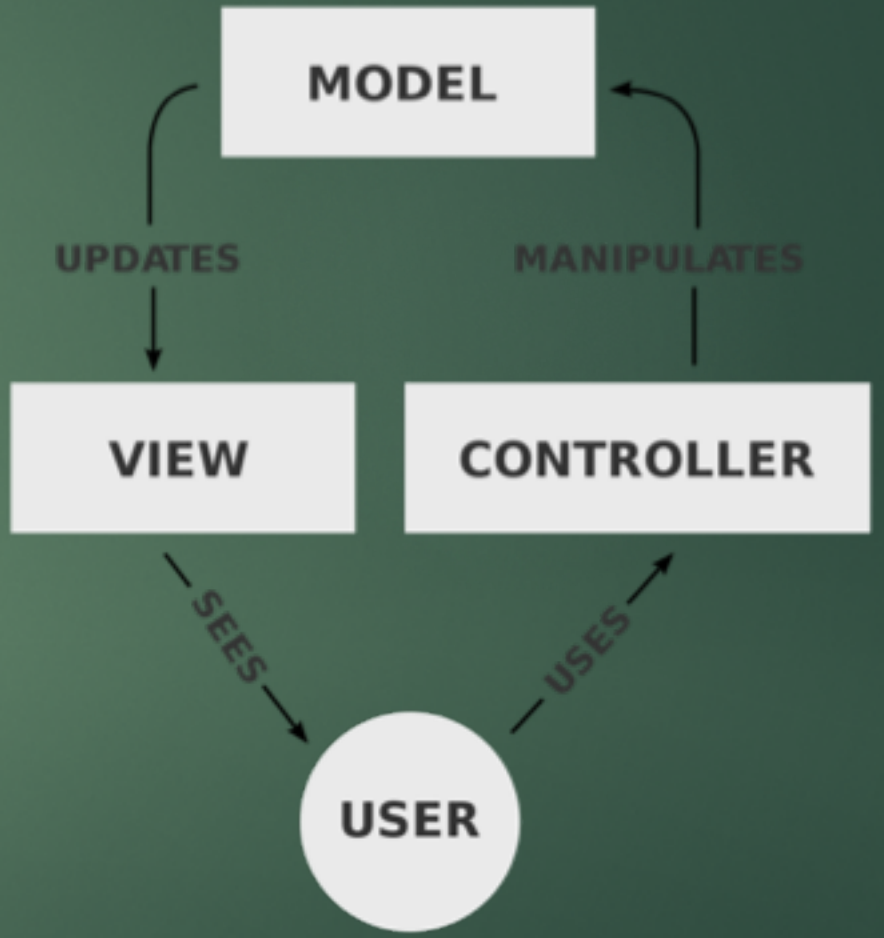
MVC : View

- All UI for the application
- Draws data from the *model* to display to user
  - Not necessarily a direct showing of the *model*
  - Data might be left out, or shown twice in different formats



MVC : Controller

- The interface (or adapter) between Model and User
- Contains logic that updates the *model* and/or *view* in response to user input
  - Can control the *view* directly if no model change is needed (e.g. change sort order)



MVC: Disadvantages

- Doesn't follow Separation of Concerns very well
  - View has references to Controller and Model
  - Controller has references to Model, and maybe View
- Doesn't follow Single Responsibility Principle very well
  - UI logic can belong to any of the three components

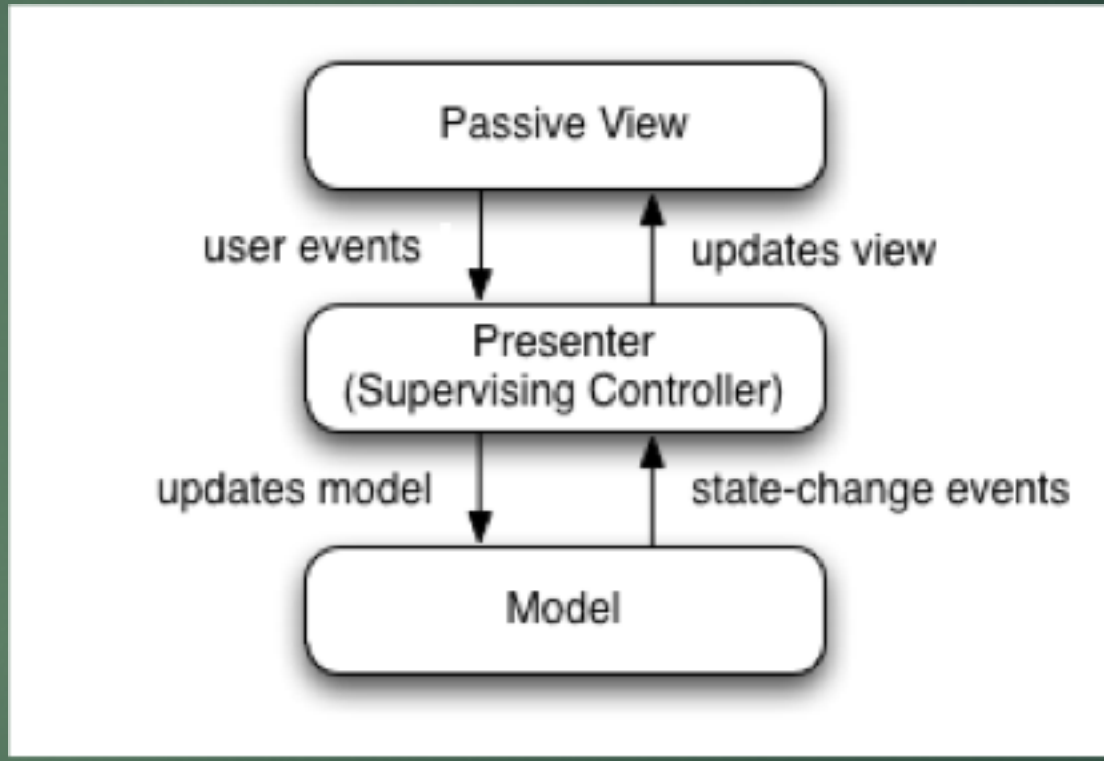
Derivative Patterns

- Model-View-Presenter (MVP)
- Model-View-ViewModel (MVVM)
- Model-View-Intent (MVI)

Model-View-Presenter

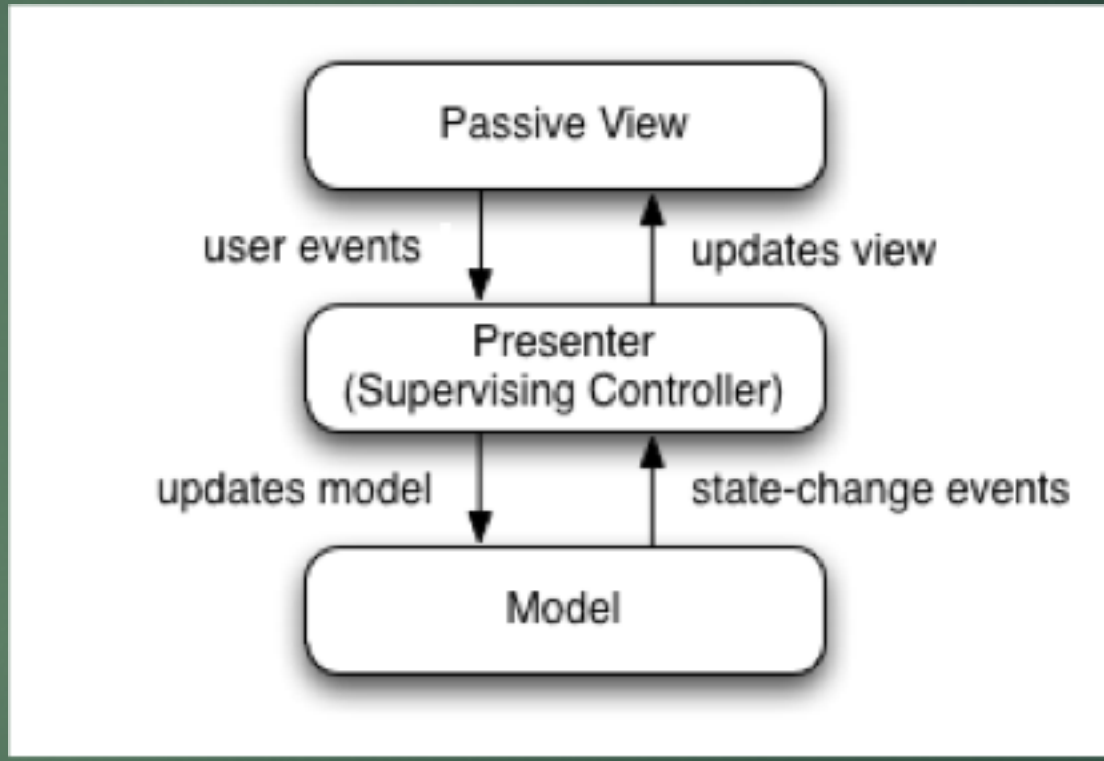
Model-View-Presenter

- Divides the application into three components:
  - The model
  - The view
  - The presenter
- Derived from MVC
- Improves separation of concerns



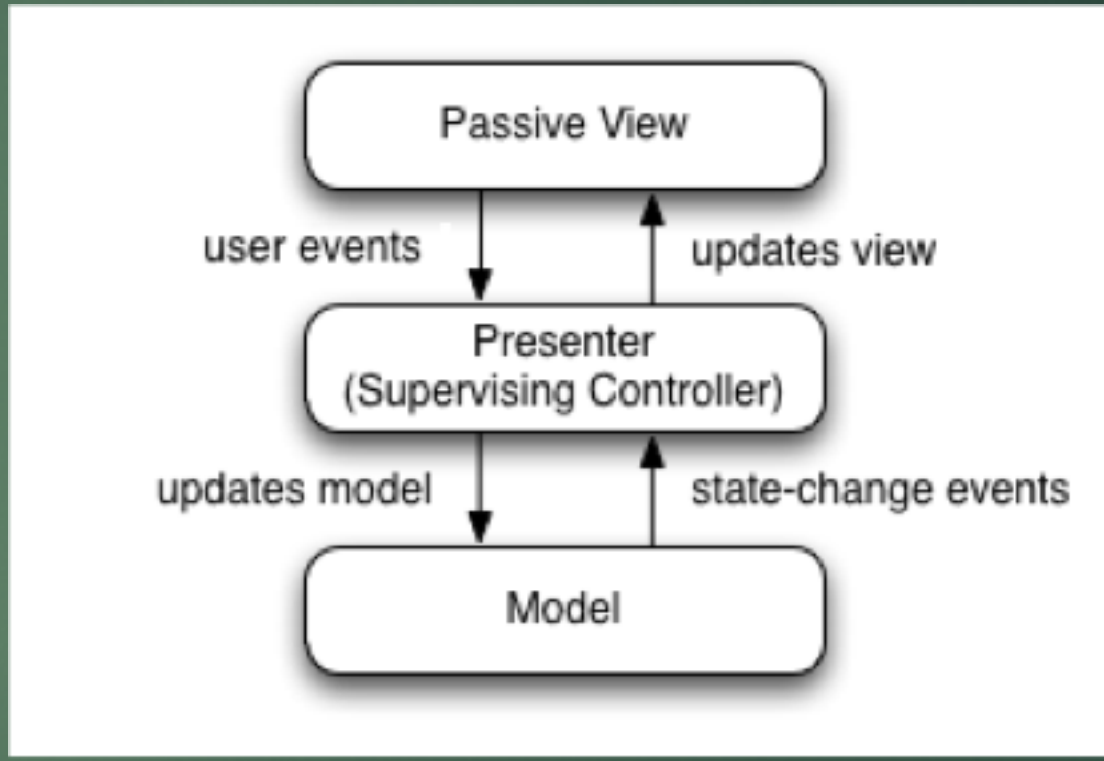
MVP: Model

- Data to be displayed in the UI
- Connected to network/database layers



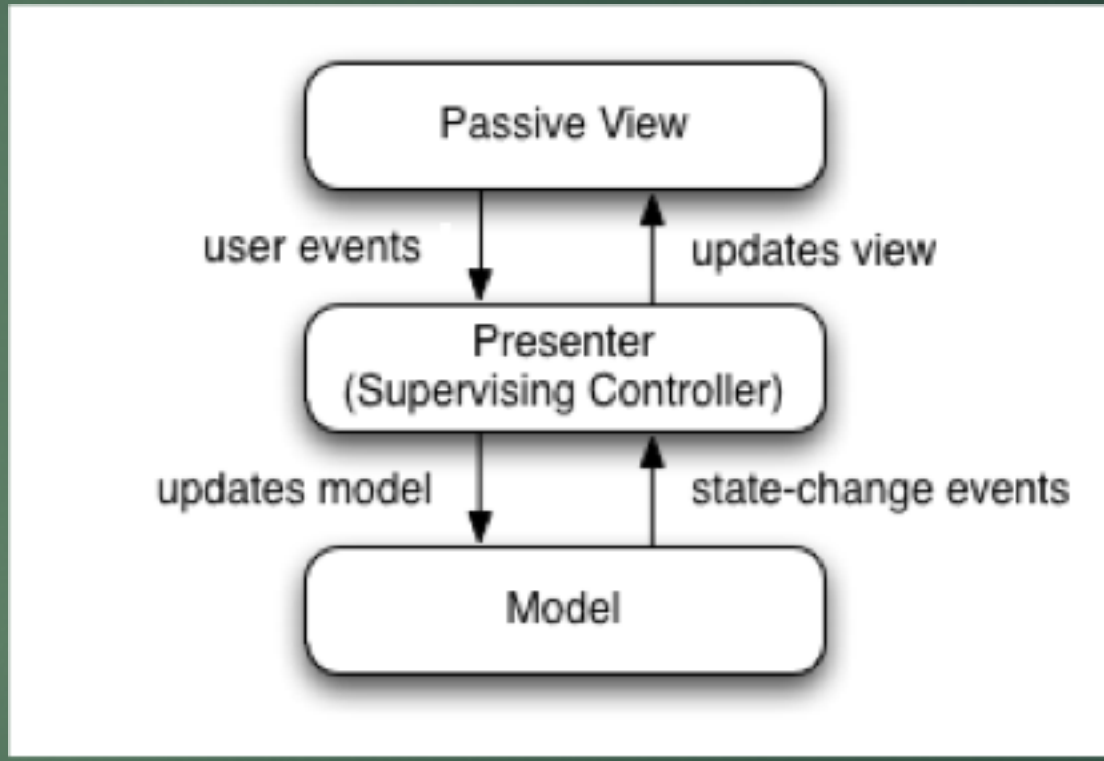
MVP: View

- Passive UI interface
- Displays data
- Routes user commands (*events*) to the presenter



MVP: Presenter

- Retrieves data from models
- Decides what to display
- Formats data for views
- Reacts to user input from the View



Model-View-Presenter

- View and Presenter work closely together
  - they need to hold a reference to each other
  - The View holds a Presenter reference, and vice versa
- Typically add a layer of abstraction between (an interface)
  - Defines connection between the two

```
// View
class RecipeActivity
        extends RecipeView {

    @Override
    displayRecipes(RecipeList) {
        // show on screen
    }
}
```

```
// View interface
interface RecipeView {
    void displayRecipes(RecipeList);
}
```

```
// Presenter
class RecipePresenter {
    private RecipeView view;

    void getRecipes() {
        // get a RecipeList model
        view.displayRecipes(list);
    }
}
```

Model-View-Presenter

- Advantages over MVC
  - Better separation of concerns
    - Breaks connection between View/Model
  - Better single responsibility across components
    - The Presenter handles all logic related to the View
    - Easier to test too
- Disadvantages
  - A lot of overhead (interfaces) for a small project
  - Presenter can become a monolithic all-knowing class (yikes!)

ModelViewViewmodel

Model-View-ViewModel

- Similar to MVP at first glance
- Better structure for an *event-driven* design
  - The Presenter told the View directly what to do
  - The ViewModel exposes streams of events for the View to bind to
- Allows quicker reactions to design changes

MVVM: Model

- Exposes data consumable through event streams
- Combines data from network, database, local storage, etc.
- Holds business logic

MVVM: View

- Active UI interface
  - Handles its own touch events
- Does not hold state
- Binds (subscribes) to event streams from the ViewModel
  - updates the screen whenever an event is received

MVVM: ViewModel

- Retrieves necessary data from Models
- Applies any necessary UI logic
- Exposes relevant properties to the View
  - May make use of intermediate objects to wrap fields
  - ex: DisplayableUser(username, email)
- Has no references to the View

Model-View-ViewModel

- View and ViewModel are loosely coupled
  - The View holds a ViewModel reference, and **not** vice versa

```
// View
class RecipeActivity {
    private ViewModel vm;

    void bind() {
        vm.getUsername()
            .subscribe(
                // your observer here:
                // update UI
            );
    }
}
```

```
// ViewModel
Class ViewModel {

    private UserDataModel data;

    Observable<String> getUsername() {
        // any processing happens here
        return data.getName();
    }
}
```

```
// Model
class UserDataModel {
    private String name;
    private String email;

    public Observable<String> getName() {
        // return, fetch from database, etc.
    }
}
```

Model-View-ViewModel

- Keeps the SoC and SRP improvements from MVP
- Less overhead than MVP
- Improved reactive programming flow
- Easy to change UI without affecting anything else